Journal of Theoretical and Applied Information Technology
20th June 2014. Vol. 64 No.2
© 2005 - 2014 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

ASSEMBLER DESIGN TECHNIQUES FOR A RECONFIGURABLE SOFT-CORE PROCESSOR

[1]SANI IRWAN MD SALIM, [2]HAMZAH ASYRANI SULAIMAN, NOR RAHIMAH JAMALUDDIN, LIZAWATI SALEHUDDIN, MUHAMMAD NOORAZLAN SHAH ZAINUDIN, YEWGUAN SOO

Faculty of Electronics and Computer Engineering, Universiti Teknikal Malaysia Melaka, Hang Tuah Jaya, 76100, Durian Tunggal, Melaka, Malaysia
E-mail: [1]sani@utem.edu.my, [2]asyrani@utem.edu.my

ABSTRACT

The reconfigurable processor design which utilizes platform such as Field Programmable Gate Array (FPGA) has offered several advantages in minimizing the non-recurring engineering cost and to reduce the time-to-market for processor-based products. However, when any modification is made to the processor architecture, the same information needs to be relayed to the assembler in order to generate the correct object files. This paper presents the assembler design techniques for a reconfigurable reduced instruction set computer (RISC) processor called UTeMRISC03. The processor is a soft-core processor, which is described in Verilog and its instruction set architecture (ISA) has been expanded to include a custom instruction. Thus, the assembler would have to adjust according to the changes being made to the ISA. One-pass and two-pass techniques have been adopted during the construction of the assembler and the generated object files are used as the initialization files during the FPGA implementation of the processor core. The assembler has been successfully developed by using both techniques and the object files are verified by executing the processor core in the Xilinx Spartan-3A FPGA chip. For comparisons, the assembling execution times for both techniques have been recorded and the one-pass technique has been able to complete the assembling process in half of the time it took for two-pass technique. The design techniques adopted in this paper will serve as the base platform with the aim of establishing a full-customizable assembler for reconfigurable soft-core processor in the near future.

Keywords: *Assembler, Reconfigurable Processor, FPGA*

1. INTRODUCTION

In the past decade, embedded system design has gone through tremendous technological advancement in every aspect of its methodologies. Conventionally, embedded systems incorporate several discrete ICs such as processor, memory chips and other I/O peripherals. A modular approach in embedded system design has given freedom to the engineers to achieve high system performance while at the same time provides easier circuit maintenance. However, the demand for smaller and compact consumer devices has led to the introduction of single chip solutions, also known as the system-on-chip methodology. By integrating all parts of the embedded system in one single die packaging, the circuit board size is significantly reduced without compromising the system performance.

The pervasiveness of embedded system applications especially in mobile and portable platforms has raised the importance of handling intensive tasks in reconfigurable devices [1]. While the embedded system could be designed to achieve the highest performance level, not all applications would require those extra gains in order to function efficiently. Furthermore, operating at the highest performance level would consume more power and directly reduced the battery life of the system. Therefore, for non-critical embedded system, the system design can be developed using logic synthesis techniques whereby the embedded processors are described using Hardware Description Language (HDL) and synthesized in a much shorter period.

The term soft-core processor refers to a processor core that is implemented using logic synthesis. To execute the processor core, the design is implemented via programmable logic devices such as Field Programmable Gate Array (FPGA). Soft-core processors are coded in HDL that enables designers to customize its internal architecture to suit the targeted application. As a comparison, a hard-core processor is a processor that is also embedded in the programmable fabric, but the architecture is fixed and cannot be modified by users. Soft-core processor offered flexibility to the designer in creating custom functional unit and to perform instruction set extension. By implementing the soft-core processor on FPGA fabric, the performance of the processor is only limited to the specification and the technology of the FPGA chip. Portability of a soft-core processor is also important as the design could easily be ported to another FPGA chip with minimal re-coding effort. Meanwhile, reduced instruction set computer (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 achieve faster instruction execution owing to its simple instruction sets that are simple and basic from which more complex instruction can be composed. Most instructions are completed in one machine cycle, which allows the processor to handle several instructions at the same time through pipelining.

In conventional processor programming flow, the assembler is tightly integrated with the processor's architecture, especially in the ISA. The ISA configuration directly affects the operation of the instruction decoder module of the soft-core processor. The issue here is when any modification is being made to the ISA, it will make the existing processor's assembler unusable due to the incompatibility of the instruction format. Furthermore, custom instruction sets, which are created in the processor, are also unavailable in the original assembler. With regards to this matter, assembler design techniques are proposed to address the assembler's compatibility issues specifically during the ISA modification procedure on a soft-core processor.

This paper will present two assembler design techniques that are applied during the development of the assembler for a reconfigurable soft-core processor. Section II discussed the related research being done on the assembler design in reconfigurable architecture. The architecture of the soft-core processor, which is utilized as the processor platform, is explained in details in Section 3. Section 4 described both one-pass and two-pass assembly techniques and their differences. Results and discussion are presented in Section 5 with the simulation results and also execution time comparison between both design techniques.

2. RELATED WORKS

Instruction program that is created by programmers, either by using low-level or high-level language, would require a compiler or an assembler in order to translate the program to a machine code. In this case, the processor's assembler is developed using various platforms which matched the processor programming procedure. The different platforms of assembler include Perl [2], eclipse plug-ins [3] and custom-made lexical and syntactical analyzer generator called GALS [4]. All of these cross-assemblers are formed as a part of an integrated design environment for their respective processor. Compiler that is developed for soft-core processors such as LEON2 [5] and Picoblaze [6] is designed to offer code optimization associated to its specific CPU architecture.

Another method that is widely adopted in developing an assembler that is based on the target architecture is the architecture description language (ADL). Essentially, ADL involved architecture modification and successively the development of the corresponding assembler. Most of the ADL tools' main objective is to simplify the processor design steps together with the supporting compilers and simulators [7]. ADL also introduced a framework that allows generation of SDK which includes compiler, assembler, simulator and debugger. All of the software tools are generated based on the processor specification and memory architectures describes by the ADL [3]. Researches have been done on system with ADL such as flat architecture description [8], retargettable compiler backend [9] and energy dissipation and monitoring [10].

3. SOFT-CORE RISC PROCESSOR

The assembler is developed based on a soft-core RISC processor architecture called UTeMRISC03 which is a 16-bit RISC processor that is described in Verilog HDL. Essentially, UTeMRISC03 has been derived from an earlier design of 8-bit RISC processor that was compatible with the popular PIC microcontroller architecture. UTeMRISC03 utilized Harvard architecture with a total of 33 instruction sets available to program the processor [11].

The modification processor from 8-bit to 16-bit processor involved significant changes being made to system busses, registers' width and other key modules such as an arithmetic logic unit (ALU) and instruction decoder. To accommodate the expanded data bus, the ISA must be modified in order to fetch a 16-bit of data from the instruction command. Simultaneously, the processor's program assembler has to be re-developed to generate the matching machine codes that suit the new ISA format. With the re-organization of ISA, there is also opportunity to customize or create new instruction sets that indirectly demonstrates the capability of implementing application specific instruction set processor (ASIP) method using a soft-core processor.

3.1 Instruction Set Architecture

The instruction register that was utilized by the original 8-bit RISC processor consisted of 12 bits. For the UTeMRISC03's ISA, the width of the IR is extended to 22 bits which comprised of 6-bits opcode, 12-bit file register address and an option of 4-bit select bit or 1-bit direction bit as shown in Figure 1. In total, there are 64 instruction sets and 4,096 file registers could be addressed according to the new ISA setting. As the processor architecture is implemented in FPGA, all modifications are made by re-programmed the Verilog code of the instruction decoder module. Design verification is done through synthesis and implementation process to ensure the modified architecture is structurally correct.

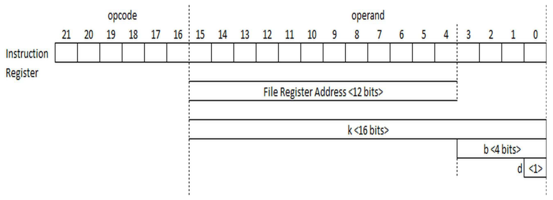


Figure 1 : Instruction Set Register Format

3.2 Custom Instruction Set Generation

Apart from the ISA expansion, a new instruction set is introduced to show the potential of UTeMRISC03 processor as a reconfigurable processor. Customized instruction sets are frequently being adopted to streamline processor operations, especially in data-intensive and complex tasks. Although instruction sets in a RISC processor are basic and simple, the processor's performance could be affected when executing complex calculation as more instruction sets are required to perform a single operation. Optimization of instructions through creating new instruction sets could reduce the total clock cycles hence improve the overall execution times.

For testing purpose, a new instruction set called 'multwf' is created in addition to the existing instruction sets. Multiplication process is executed first between two 16-bit numbers and the resulting 32-bits output is added to the pre-defined accumulator registers. A current instruction sets that is also modified to perform different function from its original setting. The 'swapn' instruction will swap the lower and upper nibble in a file register instead of swapping data from a file register to another register. The new instruction set has its own opcode and operand formatting. ALU module is also modified to perform the alternate functions once the instructions are fetched from the instruction register. Table 1 shows the list of instruction set available for UTeMRISC03 soft-core processor with shaded row indicate the new instruction set (multwf) and the modified instruction set (swapn).

Table 1: Modified Instruction Set List

Mnemonic	Opcode	Operand
swapn f,1	001110	ffff_ffff_ffff_ffff
multwf f,0	011110	ffff_ffff_ffff_ffff

4. METHODOLOGY

Fundamentally, the decoding process of the assembler utilized tokenization process and lexical analyzer in order to generate the relevant machine code from the assembly program [12, 13]. There are two design techniques proposed in this paper in order to develop a fully functional code assembler for the UTeMRISC03 soft-core processor. The implementations of both types of encoding in the assembler program model are necessary to distinguish the best approach in executing the assembler program. As the goal of the assembler is to create the object file, the execution time for the assembler to encode all the instruction sets are measured from processor file initialization to object file generation. Comparison is made on the elapsed time between the ones-pass assembler with the two-pass assembler and the result would suggest the faster encoding approach in assembler execution.

The assembler is developed in Visual Basic (VB) platform because of its complete development tools and a wide range support from users worldwide. In the code development, the assembler design applied string manipulation functions as the input and output file of the assembler are in text format. The assembler's targeted output file is called a coefficient (COE) file. The COE file is similar to hexadecimal file (.hex) which contained the machine codes of the assembly program. However, in COE file, the machine codes are arranged with COE file syntax which is designated by Xilinx.

To verify the COE file output, the soft-core processor (with the modified architecture) is implemented in the FPGA board with a memory core that is initialized with the COE file. The data in the COE file are loaded into the memory space during the memory core initialization. The soft-core processor will fetch an instruction from the memory core and decoded the command to the instruction register. Device simulation is done to observe the instruction register in order to verify that correct decoding bits are produced from the assembly program.

4.1 Two-pass Assembler Design

In two-pass assembler design, the assembling processes include the opcode/operand extraction phase (first pass) and subsequently the encoding phase (second pass) [14]. In the first pass, the assembly program file is analyzed line-by-line to identify whether each line of command contained either opcode/operands or other elements such as labels, comments and blanks. Once the opcode or operand is detected, it will be saved in a hash table together with its line number (LC) as reference. Oppositely, the process will skip to the next line when comments or blanks are identified. The special hash table is also created to store labels and its line count to be used as reference during the second pass. When the end-of-file is reached, an intermediate file called a listing file is created which contained only the line count, opcode and operand of the assembly program. The listing file is considered an organized and uncluttered version of the assembly program that omitted all unrelated elements before the encoding process.

In the second pass, the assembler will reread the listing file and directly encodes each line of instruction with the help of hash tables that are already established during the first pass. By the end of the second pass, all the machine codes are arranged and saved as the COE file. Figure 2 shows the flowchart of two-pass assembler design.

4.2 One-pass Assembler Design

In the one-pass assembler design, the instruction set encoding is executed in a single pass. After he tokenization process, each line of instruction is encoded immediately by referring to hash tables such as the symbol table and the instruction opcode table. Comment and blank lines are ignored and the process skips to the next line when both of these elements are identified. The encoded opcodes and operands are translated to their hexadecimal numbers and stored as data in the COE file. Figure 3 shows the flowchart of the implementation of one-pass assembler design.

4.2.1 Forward Referencing

The implementation of the one-pass assembler will inevitably lead to the forward referencing issue which occurred during encoding the labels. 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 used in the later part of the assembly program. In order to overcome this problem, the one-pass assembler has an additional hash table to store the labels and its corresponding line count or literal value for reference during the code encoding process.

During the tokenization process, labels are recognized either by identifying the assembler directive such as ‘EQU’ or the colon symbol which trailed the label tag at the first column of the instruction program. The labels then are stored in the symbol table which consists of a label field and a data field which could be a line count number or a literal value.

When an undefined label is detected, the label is stored in the symbol table with flag ‘X’ is marked as its data field indicating that the label is yet to be defined. The flag ‘X’ will be overwritten with correct line count or literal value once the same label is identified in the subsequent instructions. The assembler also will keep track all of the undefined labels and constantly updating its entries until all of the label’s data are properly addressed. By the end-of-file, the symbol table’s entries are supposed to be clear without any ‘X’ flag in the data field or else an error message would be generated implying that there are unrecognized labels that are yet to be addressed.

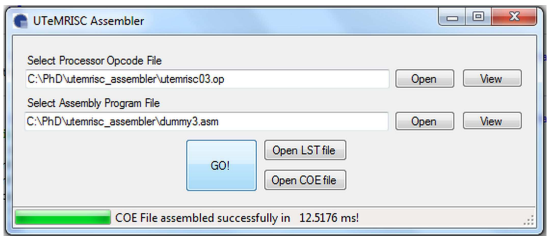
4.3 FPGA Implementation

The UTeMRISC03 processor is setup in the Xilinx ISE Design Suite for FPGA implementation procedures which include modules’ initialization, floor planning and timing constraint setup. To store all the program instruction in ROM, the COREgen section is invoked to instantiate the single block memory module with 2048 x 22 bits in size. During this process, the COE file is loaded as a memory initialization file.

Once the machine code resides in the ROM module, the UTeMRISC03 processor design is implemented using the common FPGA design flow. Initially, a behavioral simulation process is conducted in order to verify the syntax and the functionality of the processor's architecture without any timing information. The Verilog design files then are synthesized, translated, mapped and placed-and- routed to generate a netlist file called bitstream. The bitstream file is programmed into the Spartan-3AN FPGA board using a JTAG programmer. To observe the content of the processor's internal registers during the FPGA execution, a behavioral simulation process is conducted through the ISim The selected internal signals can be observed through the ISim waveform window to verify the assembly program sequence, instruction decoding and the instruction command execution.

5. RESULTS AND DISCUSSION

The assembler is built using the Visual Basic platform and Figure 4 shows the graphical user interface for the assembler. The interface is made with simple and clear layout for the users to easily select the appropriate files and review the output files once the assembling process is completed.



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

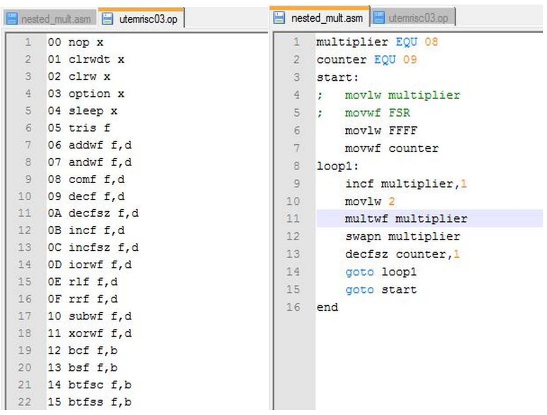
Select Assembly Program File
C:\PhD\utemrisc_assembler\dummy3.asm

COE File assembled successfully in 12.5176 ms!

Figure 4: Graphical User Interface of the Assembler

Firstly, the user is required to select a processor opcode file with the extension .op. The processor opcode file contained all the instruction sets' mnemonic, opcode and its instruction set types. The information in this file is stored in a hash table and for references during the tokenization and lexical analysis process. The instruction set types will indicate the number of operands expected for the specified instruction set. Hence, the lexical analyzer will ensure that the correct operands are existed and matched the instruction set types. The translation of the instruction mnemonics to its respective opcode are also referred to the processor opcode file.

The assembly program file is also loaded to the assembler. The assembly program file, with extension .asm, is a file that contains the source code of a program which is written in assembly language. The instruction set used in the assembly program must adhere to the instruction set list as mentioned in Table 1. Users can examine the content of both processor opcode and assembly program files in Notepad application by clicking the 'View' button. Both opcode file and assembly program file are shown in Figure 5.

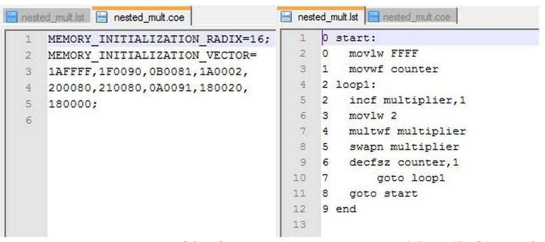


```
00 nop x
01 clrwdt x
02 clrw x
03 option x
04 sleep x
05 tris f
06 addwf f,d
07 andwf f,d
08 comf f,d
09 decf f,d
0A decfsz f,d
0B incf f,d
0C incfsz f,d
0D iorwf f,d
0E rlf f,d
0F rrf f,d
10 subwf f,d
11 xorwf f,d
12 bcf f,b
13 bsf f,b
14 btfsc f,b
15 btfss f,b
```

```
multiplier EQU 08
counter EQU 09
start:
;   movlw multiplier
;   movwf FSR
    movlw FFFF
    movwf counter
loop1:
    incf multiplier,1
    movlw 2
    multwf multiplier
    swapn multiplier
    decfsz counter,1
    goto loop1
    goto start
end
```

Figure 5: Assembly Program File (left); Processor Opcode File (right)

Button 'Go' is pressed in order to start the assembling process. Both processor opcode file and assembly program file are simultaneously read and pass through tokenization and lexical analysis process. A status bar indicator at the bottom of the interface would indicate the progress during the execution of the processes. The elapsed time for the assembler operation is displayed in the status bar once all of the processes have been successfully completed. The elapsed time is measured from the start of the input files initiation until the generation of the COE file. Figure 6 shows the output files of both the listing file (in two-pass assembler) and the COE file. To reflect the accuracy of the elapsed time, the assembling process is repeated several times and the average time is calculated. Table 2 shows the results of the execution time of the assembling process on the designated assembly program file.



```
MEMORY_INITIALIZATION_RADIX=16;
MEMORY_INITIALIZATION_VECTOR=
1AFFFF,1F0090,0B0081,1A0002,
200080,210080,0A0091,180020,
180000;
```

```
0 start:
0   movlw FFFF
1   movwf counter
2 loop1:
2   incf multiplier,1
3   movlw 2
4   multwf multiplier
5   swapn multiplier
6   decfsz counter,1
7       goto loop1
8   goto start
9 end
```

Figure 6: Listing file for Two-Pass Assembler (left) and the COE File (right)

Table 2: Results of the Average Execution Times

Assembler Design Technique	Average Execution Times
One-pass	1.16 ms
Two-pass	2.22 ms

To observe the generated COE file, the 'Open COE file' button is pressed and the COE file will be opened through the Notepad application. User can compare and cross check the COE file output with the listing file output to verify its accuracy. Essentially, both files contained the same data set of opcode but in different formatting. Although the assembler execution is done on both one-pass and two-pass design techniques, the generated COE file of both techniques is exactly the same. This is because the assembler took the same input files, hence the assembling processes are done to the same assembly program and using the same instruction set list.

Subsequently, in the FPGA implementation, the COE file is loaded into the memory module during the ROM generation inside the UTeMRISC03 architecture. After a successful synthesis process, the UTeMRISC03 architecture is translated, mapped and placed-and-routed before being programmed in the FPGA. Figure 7 shows the waveforms of selected internal signals observed during the behavioral simulation of the UTeMRISC03 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 the instructions correctly and in a timely manner.

As shown in Figure 7, the test program is implemented by the RISC processor core is in a correct sequence as defined in the coefficient file. All instructions are executed in a single clock cycle with the exception of 'goto' instruction which required two clock cycles (GOTO and NOP instruction). The assembler has successfully assembled the assembly language program to its equivalent hexadecimal format which is readable by the RISC processor core. The black circles in the Figure 7 indicate the new instruction that is created during the custom instruction set generation phase. The 'goto' instruction is also functioning properly as the PC register is constantly updated in accordance with the program flow.

With regards to the inner working operation of the UTeMRISC03 core, the instructions are fetched from the ROM address pointed by the PC register. Then, the machine code is decoded in order to identify its opcode and operand. The instruction decoding process is performed by the decoder module which is heavily reliant to the instruction set architecture. As a consequence, both instruction opcode file and instruction decoder module in the UTeMRISC03 core must have an identical set of instruction opcode in order to successfully decode and assemble any test program during the FPGA implementation. After the operands are identified, whether it is a literal value, file register address or direct memory address, the operands are fed to the ALU module for command execution. The output results are observed as internal signals and could be sourced out to the input/output port depending on the processor core design.

From the execution time recorded for both one-pass assembler and two-pass assembler, it is clear that one-pass approach to the assembler design yielded the fastest execution time in generating the listing file and the coefficient file. The obvious advantage of one-pass assembler is the ability to process the assembly program file in single-read procedure that ultimately halved the execution time required by the two-pass assembler. The forward referencing issue is solved by using additional hash tables that flagged any of the yet-to-be-defined symbols. Incidentally, the resource requirement for one-pass assembler execution is significantly higher due to more allocation is needed to generate the required hash tables.

On the other hand, two-pass assembler offered more organized and simpler assembling process, albeit delays in the execution time. 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, the longer execution time is contributed by the multiple read/write processes on the listing file and also to generate the coefficient file towards the end of the process.

6. CONCLUSION

A customized assembler is an important component when designing a RISC processor core using the ASIP design methodology. The modification of the processor's ISA that is tailor-made for a specific application would directly improve certain aspects of the processor's performance. However, the ISA modification would require a compatible assembler and also an instruction decoder module with identical data set of instruction opcodes. The assembler presented in this paper has the ability to adopt the changes made in the ISA and also to include new instruction set through the processor opcode file. One-pass and two-pass design techniques have been adopted during the assembler development and one-pass assembler provides the quickest assembling time in order to generate the coefficient file. The coefficient file is loaded in ROM module into the UTeMRISC03 core and the implementation on the FPGA chip has produced correct program execution of the assembly program file. Overall, the design techniques adopted in the assembler development would be a good starting point for future development of a customizable and full-scale compiler for an ASIP processor design.

ACKNOLEDGEMANT:

This work was supported by the Universiti Teknikal Malaysia Melaka and Ministry of Education Malaysia through the research grant number PJP/2012/FKEKK(45C)/S01050.

REFRENCES:

[1] I. Skliarova, T. Vallejo, V. Sklyarov, A. Sudnitson, and M. Kruus, "Solving Computationally Intensive Problems in Reconfigurable Hardware: A Case Study", *Journal of Convergence Information Technology,* vol. 8, 2013.

[2] K. Nakano, K. Kawakami, K. Shigemoto, Y. Kamada, and Y. Ito, "A Tiny Processing System for Education and Small Embedded Systems on the FPGAs", in *IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, EUC '08* 2008, pp. 472-479.

[3] 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.

[4] L. Taglietti, J. O. C. Filho, D. C. Casarotto, O. J. V. Furtado, and L. C. V. dos Santos, "Automatically retargetable pre-processor and assembler generation for ASIPs", in *The 3rd International IEEE-NEWCAS Conference*, 2005, pp. 215-218.

[5] Y. Li, X. Zhu, W. Zhang, C. Wang, and Z. Deng, "Research and design of CPU for teaching based on SPARC V8", *Journal of Theoretical and Applied Information Technology,* vol. 50, pp. 352-357, 2013.

[6] A. Z. Mansoor, M. R. Khalil, and O. A. Jasim, "Position control of DC servo motors using soft-core processor on FPGA to move robot arm", *Journal of Theoretical and Applied Information Technology,* vol. 32, pp. 99-106, 2011.

[7] P. Karlstrom, S. Loganathan, F. Akhlaq, and D. Liu, "Automatic assembler generator for NoGap", in *Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)* 2010, pp. 1-4.

[8] L. Ghica, B. Ditu, and N. Tapus, "Automatic Generation of Architecture Model for Reconfigurable Build Tools", in *19th International Conference on Control Systems and Computer Science (CSCS)*, 2013, pp. 142-146.

[9] 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.

[10] K. Ko, "ADL-Driven Simulator Generation for Energy Dissipation Tracing and Monitoring," in *Future Information Technology, Application, and Service*. vol. 164, J. J. Park, V. C. M. Leung, C.-L. Wang, and T. Shon, Eds., ed: Springer Netherlands, 2012, pp. 459-466.

[11] A. J. Salim, S. I. M. Salim, N. R. Samsudin, and Y. Soo, "Instruction Set Extension Through Partial Customization of Low-End RISC Processor", *Australian Journal of Basic and Applied Sciences,* vol. 7, pp. 678-687, 2013.

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

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

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

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

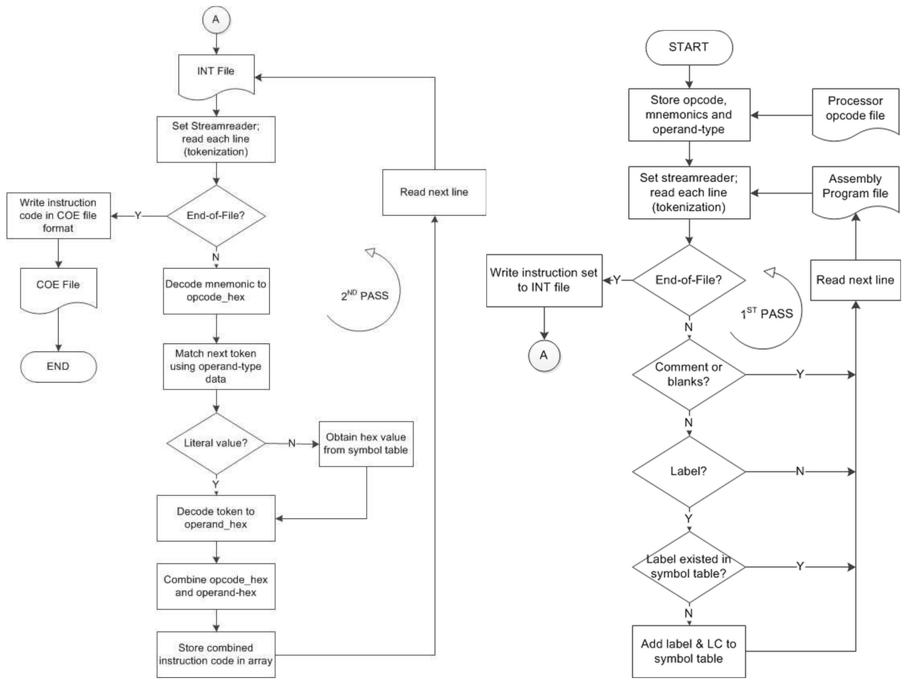

A
INT File
Set Streamreader; read each line (tokenization)
End-of-File?
Y
Write instruction code in COE file format
COE File
END
N
Decode mnemonic to opcode_hex
Match next token using operand-type data
Literal value?
N
Obtain hex value from symbol table
Y
Decode token to operand_hex
Combine opcode_hex and operand-hex
Store combined instruction code in array
Read next line
2ND PASS
START
Store opcode, mnemonics and operand-type
Processor opcode file
Set streamreader; read each line (tokenization)
Assembly Program file
End-of-File?
Y
Write instruction set to INT file
A
1ST PASS
Read next line
N
Comment or blanks?
Y
N
Label?
N
Y
Label existed in symbol table?
Y
N
Add label & LC to symbol table

Figure 2: Flowchart of the Two-Pass Assembler Design

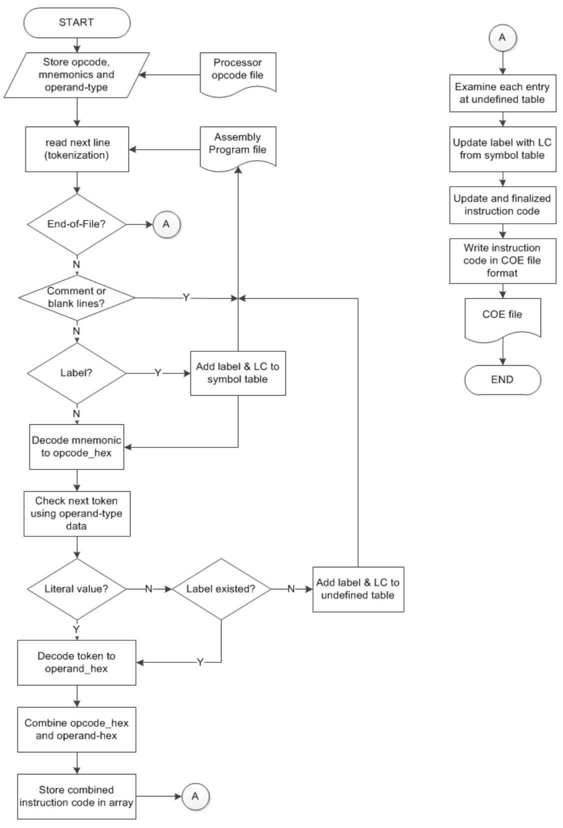

START
Store opcode, mnemonics and operand-type
Processor opcode file
read next line (tokenization)
Assembly Program file
End-of-File?
A
N
Comment or blank lines?
Y
N
Label?
Y
Add label & LC to symbol table
N
Decode mnemonic to opcode_hex
Check next token using operand-type data
Literal value?
N
Label existed?
N
Add label & LC to undefined table
Y
Y
Decode token to operand_hex
Combine opcode_hex and operand-hex
Store combined instruction code in array
A
A
Examine each entry at undefined table
Update label with LC from symbol table
Update and finalized instruction code
Write instruction code in COE file format
COE file
END

Figure 3: Flowchart of the One-Pass Assembler Design

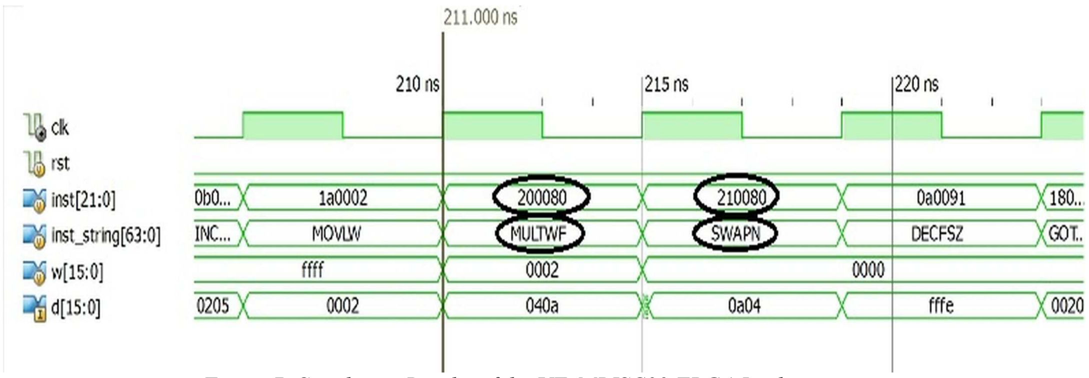


Figure 7: Simulation Results of the UTeMRISC03 FPGA Implementation