디지털산업정보학회 논문지 제11권 제3호-2015년 9월
http://dx.doi.org/10.17662/ksdim.2015.11.3.031

고급 언어에서 ASIP을 위한 전용 부호 생성 기술 연구*

알람 삼술** · 최 광 석***

A Custom Code Generation Technique for ASIPs from High-level Language

Alam S. M. Shamsul · Choi Goangseog

〈Abstract〉

In this paper, we discuss a code generation technique for custom transport triggered architecture (TTA) from a high-level language structure. This methodology is implemented by using TTA-based Co-design Environment (TCE) tool. The results show how the scheduler exploits instruction level parallelism in the custom target architecture and source program. Thus, the scheduler generates parallel TTA instructions using lower cycle counts than the sequential scheduling algorithm. Moreover, we take Tensilica tool to make a comparison with TCE. Because of the efficiency of TTA, TCE takes less execution cycles compared to Tensilica configurations. Finally, this paper shows that it requires only 7 cycles to generate the parallel TTA instruction set for implementing Cyclic Redundancy Check (CRC) applications as an input design, and presents the code generation technique to move complexity from the processor software to hardware architecture. This method can be applicable lots of channel Codecs like CRC and source Codecs like High Efficiency Video Coding (HEVC).

Key Words : Transport Triggered Architecture(TTA), TTA-based Co-design Environment (TCE), Instruction Level Parallelism (ILP), Architecture Definition File(ADF), Implementation Definition File(IDF)

Ⅰ. Introduction

To allow easy customization of processor design, TTA is one of the most suitable ASIP architecture templates. To improve the processor speed, concurrent execution of instructions known as Instruction Level Parallelism (ILP) is very important. This is an attractive approach to satisfy high performance requirements. Due to the flexibility and scalability behavior of TTA architecture, it is an interesting choice for the design of ASIPs. TTAs are constructed from multiple, concurrently-operating function units (FUs), and each FU supports RISC-style operations. That means a TTA processor does not need to

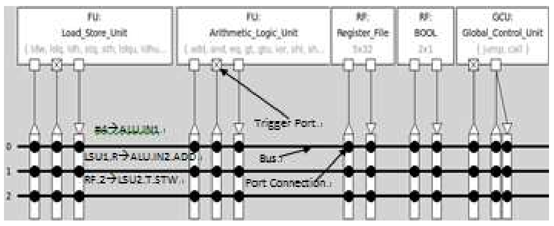
* 이 논문은 2015년 조선대학교 연구비 지원을 받아 연구되었음.
** Khulna대학교 전자통신공학과 조교수
*** 조선대학교 정보통신공학과 교수(교신저자)

include complex instruction dependency detection hardware logic, which simplifies the processor implementation [1].

In TTA, operations are started as side effects of writing operand data to the trigger port of the FU. These FUs are internally pipe-lined, and it is possible to implement one or more operations using FUs. When the operation execution is triggered, the result can be read from the output port after the time defined by the static latency of the operation.

Figure 1 shows an example of a TTA processor data-path that consists of FUs, register files (RFs), a Boolean RF, and a custom interconnected network [2]. These data transports are clearly programmed and written to a trigger port of functional units. Figure 1 also represents instructions, defined as moves, for three buses [2]. An explanation of these instructions is given in the next section. In this figure, moves are defined for three buses performing an integer summation loaded from memory and a constant. Besides the code generation technique using TCE, a comparison between TCE and Tensilica tools is displayed in terms of cycle count. At first, we will discuss an ASIP oriented design flow using XtensaXplorer (XX) integrated development environment (IDE) as the design frame work under Tensilica tool. Then, we will explain the code generation techniques using TCE, and finally the comparison result between TCE and IDE will be portrayed.

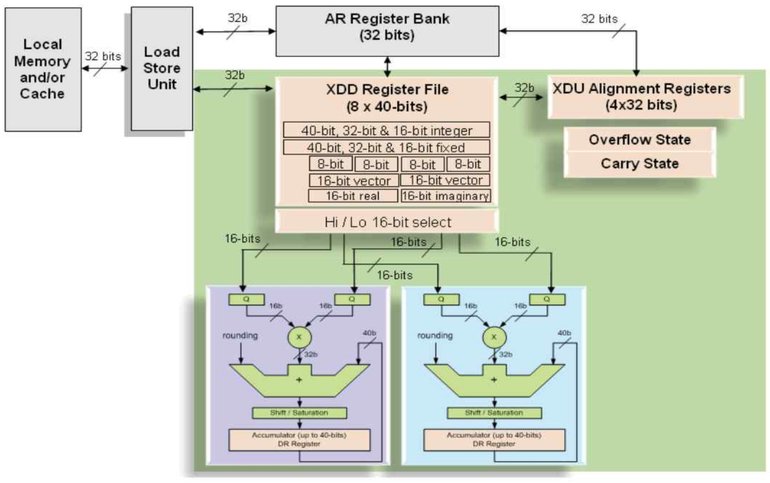
Using the XX, it is possible to integrate software development, processor optimization and multiple-processor system-on chip (SoC) architecture into one common platform. From it, we can profile



<Fig. 1> Example of TTA processor data path with 3 instructions for three buses

our input application code to identify the cycle consumed by the function used in input design. Then, we can make necessary change to speed up that code. There are various building blocks in the Xtensa architecture. The system designer should specify the different blocks of configuration function units. Advanced designer-defined functions are one kind of hardware execution units and registers. Based on these properties of this architecture, we have taken different configurations of architectures to simulate our input application. As mentioned earlier, beside the TTA code generation of CRC 32, we are going to define several experiments to find a good XX processor tuned to this application. For this reason, we have taken 16 preconfigured cores and the result is tabulated after simulating the input application using those cores. Then, we apply some custom logic levels to processor for accelerating the processor performance. These preconfigured cores are divided into four broad categories; Communication, HiFi/Audio, Video/Imaging and Diamond or General Purpose Controller.

Recently, Tensilica introduced the high-performance, small, low-power 16-bit dual-MAC (Multiply-Accumulate) Digital Signal


Local Memory and/or Cache
32 bits
Load Store Unit
32b
AR Register Bank (32 bits)
32 bits
XDD Register File (8 x 40-bits)
40-bit, 32-bit & 16-bit integer
40-bit, 32-bit & 16-bit fixed
8-bit
16-bit vector
16-bit real
16-bit imaginary
Hi / Lo 16-bit select
XDU Alignment Registers (4x32 bits)
Overflow State
Carry State
16-bits
rounding
Shift / Saturation
Accumulator (up to 40-bits) DR Register

<Fig. 2> A simplified architecture of ConnXD2 DSP Engine [7]

Processor (DSP) engine. This communication configuration core is known as ConnX D2DSP engine[7]. In this paper, two ConnX configurations known as XRC_D2MR and XRC_D2SA are used for simulation and show very good performance between all other configurations. The XRC_D2xx configuration includes dual 16-bit MAC units and 40-bit register file to the base RISC architecture of the Xtensa LX processor. This engine uses two-way Single Instruction Multiple Data (SIMD) instructions to provide high performance on vectorizable C code. It implements an improved form of Very Long Instruction Word (VLIW) instructions and five-stage pipeline.

Figure 2 shows the basic architecture of the ConnX D2 engine with two MAC units with register banks [7]. The ConnX D2 instruction set is designed for numeric computations like add-subtract, add-compare or add-modulo etc required for digital signal processing. This ConnX D2 core exploits seven DSP-centric addressing scheme mentioned in figure 2. In order to provide excellent performance, it includes data manipulation instructions like shifting, swapping, and logical operations. As mentioned before, the input design is CRC 32 and it has huge number of shifting, swapping and logical operations. So, this processor architecture is suitable for the input design.

For SoC design, Xtensa LX processor provides the I/O bandwidth, compute parallelism, and low-power optimization equivalent to hand-optimized, register transfer level (RTL)-designed non-programmable hardware blocks. The HiFi/Audio engine (330HiFi) is optimized for audio processor, voice codecs and pre- and post-processing modules. This configuration includes the Xtensa LX processor that is the basis of the 330HiFi processor. It extends the HiFi 2 Audio Engine instruction set architecture (ISA) for hardware perfecting, 32x24 bit multiply/accumulate operations, circular buffer loads and stores and bidirectional shift. There are two main components in this engine: a DSP subsystem that operates primarily on 24-bit data items and other one is a subsystem to assist with bit stream access and variable length encoding and decoding [9]. So this architecture is fully compatible for audio/video compression or processing operation.

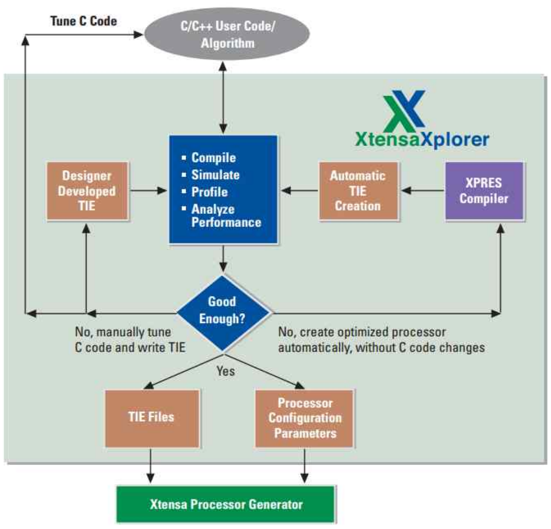
Another category of processor known as Diamond or General Purpose Controllers are optimized for SoC design and it can be used in any application where a controller is required. Diamond controllers are based on a modern RISC architecture. Among these controllers, Diamond 106Micro and 108Mini are cache-less controllers and designed for lowest area and power. The Diamond 106Micro has an iterative, multi-cycle multiplier and uses a non-windowed 16-entry address register (AR) file. So it is ideal for fast context switching and does better performance for nested function calls. The diamond 108Mini has full 32x32 multiplier and divider and 32-bit input and output general-purpose I/O (GPIO) ports. The Diamond 212GP and 233L are applicable for medium level performance and they have caches, local memories, divider, 32-bit input/output GPIO ports and other DSP instructions. Therefore, Diamond 212GP and 33L are ideal for hard drive controller, imaging, printing, networking etc. The Diamond 570T can generate up to 64-bit VLIW instruction bundles as per the requirement of input design. This VLIW instruction contains two or three operations or instructions. The 570T processor also includes 32-bit input and output GPIO ports with 32-bit input and output First-In First-Out (FIFO) interface. Therefore, this FIFO interface provides a very useful mechanism for the processor to communicate with other RTL blocks, devices and processors [9]. Next, we will show the comparative performance of all these processor architecture.

To accelerate the speed of the processor in Tensilica, it is possible to apply the custom operation in input design. Tensilica Instruction Extension (TIE) language is a powerful way to optimize the processor and is used to describe new instructions, new registers and execution units that are automatically added to the Xtensa processor. The processor take TIE files as input and creates a version of Xtensa processor to complete the tool chain incorporate with new TIE instruction.

Figure 3 shows the TIE generation technique using Xtensa processor. This TIE can be generated automatically or manually, depending on the performance of TIE instructions. In this paper, we have used TIE instructions generated automatically to profile our input design and it shows good performance. So using TIE instruction, processor creates single instructions that perform the multiple general purpose instruction.

As mentioned above, TIE instructions improve the execution speed of the input application running on Xtensa processor. Some other techniques like Flexible Instruction Extensions(FLIX), and SIMD can be executable through TIE operation. In this paper, we applied only FLIX instruction to the input application.

In Xtensa, FLIX instructions are multi-operation instructions (32-bit or 64-bit long) that allow a


Tune C Code
C/C++ User Code/ Algorithm
XtensaXplorer
Designer Developed TIE
• Compile
• Simulate
• Profile
• Analyze Performance
Automatic TIE Creation
XPRES Compiler
Good Enough?
No, manually tune C code and write TIE
No, create optimized processor automatically, without C code changes
Yes
TIE Files
Processor Configuration Parameters
Xtensa Processor Generator

<Fig. 3> Generation of custom TIE instructions [10]

processor to perform multiple, simultaneous, independent operations. In FLIX, processors are encoding the multiple operations into a wide instruction word. The Xtensa C compiler (XCC) takes the FLIX operation and converts it into FLIX format instruction as per the requirements to accelerate the input code [7]. The performance of FLIX instruction will be discussed in simulation result section.

Ⅱ. TTA Programming

In TTA programming, data transports are required to read and write the operand values, and the operation is triggered when data is written to a trigger port. Sequential and parallel TTA programs represent the sequence of instructions depending on a number of buses. In sequential TTA programming, the moves are sequentially executed because of single bus architecture. Therefore, its code is not scheduled to be executed in a target structure. In a parallel TTA program, a set of moves is executed using a multiple bus structure. Therefore, each bus will be utilized in parallel in the same clock cycle. Thus, ILP is exploited in a parallel TTA architecture. An example of a simple TTA program is given below [3]:

```
1: 100 ->RF. 1 ; 500 -> RF. 2
2: RF. 1 -> ALU. add. 1; RF. 2 -> ALU. sub. 1
3: 50 ->ALU. add. 2 ; 100-> ALU. sub. 2
4: ALU. add. 3 ->RF. 1 ; ALU. sub. 3 ->RF. 2
5: RF. 1 ->ALU. EQ. 1 ; RF. 2 -> ALU. EQ. 2
6: !ALU. EQ. 3->bool; ········
7: !bool 2-> GCU. jamp. 1
```

In here, two buses are used in TTA architecture so that a couple of instructions are executed in one clock cycle. In Line 1, two general-purpose registers take constant values from the immediate unit and store those values in the ADD (addition) and SUB (subtraction) modules of Arithmetic Logic Unit (ALU) through a load store unit (LSU). This is explained in Line 2. After finishing the similar operations in Lines 3 and 4, RF_1 and RF_2 hold the output values of ADD and the SUB module of ALU. Line 5 shows that these two values from GPRs are applied to two inputs of the equator (EQ) module of ALU. In Line 6, the result of the comparison is transferred to a Boolean register, which is used in conditional execution. In the last line, the value of the Boolean register is evaluated and the jump operation of the global control unit (GCU) is triggered in case a Boolean register value is false. That means the program execution is transferred back to Line2 when the values of RF_1 and RF_2 are not equal. For this example, the second operand of the ADD, SUB, and EQ operations, and the first operation of the JUMP operation, are triggering ports. Therefore, this whole comparison operation is done in 7cycles, and each cycle executes two operations for two bus architectures. That means, depending on this ILP, the speed of the processor is identified. Single bus architecture would require almost 12 cycles to execute this operation. The assembly notations of this example are taken from the TCE tool [3].

In TTA architecture, it is possible to add a new instruction to the target processor which implements arbitrary functionality. This custom instruction reduces longer chain operations to a single custom operation. To add this custom instruction, the ADF files of the TTA processor should be modified by introducing a new FU. In this paper, we showed the ways in which the instructions set are generated from each custom function unit. The generating procedure of each efficient custom function unit, modification of ADFs, and reference design are discussed in the author's other paper [4]. The TTA code generation techniques for the custom architecture named ascrcfast. adf, are discussed in detail. Moreover, this new custom architecture for implementing CRC is very efficient in terms of cycle count which is also discussed in-depth in [4].

Ⅲ. Code Generation Method using TCE Tool

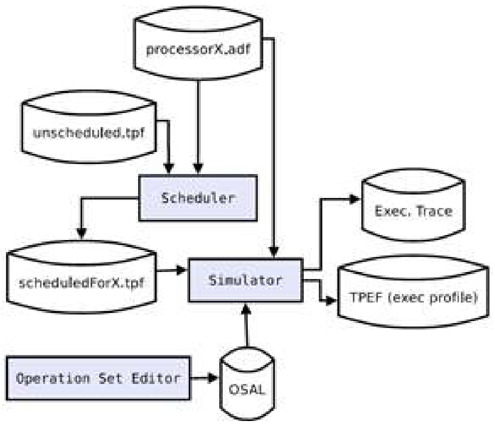
In the previous section, we discussed the assembly instruction of the TTA processor, which was applied to ADFs in the TCE tool [6]. In this section, we will discuss the code generation technique which is the main part of whole design flow in the TCE structure. Before going to discuss the code generation technique using TCE tool, we will show the advantage of customized code generation for TTAs. It is well-known that VLIW and TTA based processors exploit the ILP at compile time. Here, compiler finds the parallel instructions before run time. VLIWs are constructed from multiple, concurrently operating FUs where each FU supports RISC style operation. But the traditional VLIW processor architecture is not suitable for scalable operation because of its complex connectivity of required data-path especially for register file and bypass circuit. The data bandwidth and instruction bandwidth depend on the number of selected FUs. However, when all FUs are utilized, the available data bandwidth is still rarely utilized. For that reason, the concept of TTA and its code generation techniques are required.

The complete design flow is divided into four phases: Initialization, Design Space Exploration, Code Generation, and Processor & Program Image Generation [3].

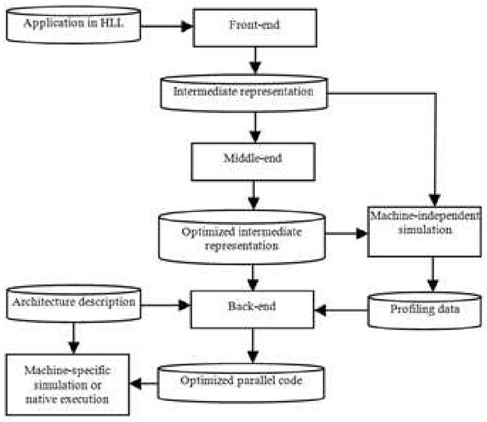
In the initialization phase, the sequential code form of the TTA Program Exchange Format (TPEF) is generated by compiler like TCECC (TCE C Compiler) including the ADF. If this compiler is provided with multiple compilation units, the TPEF linker links them to a single TPEF binary file. This TPEF file format is used for storing unscheduled, partially scheduled, and scheduled TTA programs to apply input to TCE. The compiler used here is known as a frontend compiler because it has no more use in the rest of the TCE toolset. Now, for TCE version 1.5, this compiler can compile only in the high level C language.

Design space exploration is used to estimate the cost for different starting point architectures. The goal of this phase is to find an optimal architecture for input design. Here the explorer removes the unused connections and resources from the starting point architecture, which is more beneficial in terms of area, power, and time. It should be noted that if a program is simulated using various types of efficient target architecture modified either automatically or manually, parallel simulation is invoked to increase processor speed. So, the Explorer creates a database named the Exploration Result Database (ExpResDB), which contains the configuration of evaluations during exploration. It also creates an Implementation Definition File (IDF) for estimating the cost of explored target architecture.

The most influential and demanding part of TCE design flow is code generation and analysis.



<Fig. 4> Code generation and analysis [3]

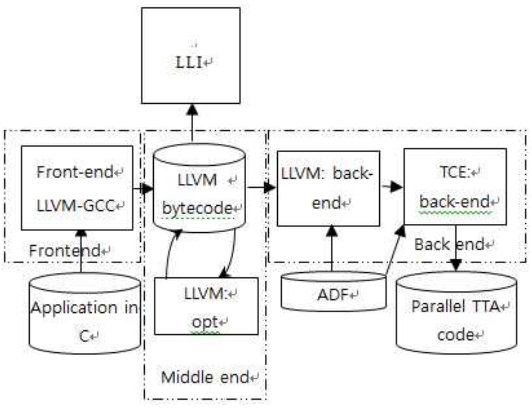

Application in HLL
Front-end
Intermediate representation
Middle-end
Optimized intermediate representation
Machine-independent simulation
Architecture description
Back-end
Profiling data
Machine-specific simulation or native execution
Optimized parallel code

<Fig. 5> Data flow in the ILP compiler [5]

Figure 4 shows the code generation procedure of the TCE tool. In this stage, the sequential program is converted to parallel instructions by efficiently utilizing the given target architecture. It is very difficult for a programmer to write a thousand lines of TTA program manually, even if there is a use of semi-automatic design space exploration. Moreover, hand written code is not always efficient. Therefore, in this stage, the scheduler takes all responsibility for the performance of the entire toolset [3]. Figure 5shows the important concepts regarding an instruction scheduling compiler for the TTA architecture. Generally, the main working principle of a compiler is to translate a program written in a source language to another target language.

In TCE, the compiler is used to translate high level language (HLL) like C into executable code for TTA. It should be noted that, during this compilation, it assigns processor resources to every data transport, while avoiding any conflicts in resource usage [5]. Moreover, at the same time, all possible ILP should be exploited to facilitate efficient code execution. Figure 5 show that an ILP compiler has three parts: a front-end, a middle-end, and a back-end [5].

The front-end translates the source application code written in HLL into intermediate program representation (IR), and this IR is not compiled for any particular target architecture. All possible auxiliary data, including IR, is the input to the middle-end of compiler (or back-end if there is no optimization performed on IR). The middle-end executes high-level language and architecture independent optimization on IR produced by the front-end. To increase efficient ILP, this optimization includes dead-code elimination, function in-lining, and loop unrolling. In the back-end, the compiler reads machine-independent IR, the ADF, and profiling information. Then it translates the code into parallel code for the target architecture. The back-end performs several optimizations using control analysis, data flow analysis, and memory reference disambiguation analysis. These optimizations comprise register allocation and instruction scheduling, which are important parts of generating efficient code executables for the target processor [5].



< Fig. 6> Structure and data flow in a TCE compiler [5]

Figure 6 shows the basic structure of the TCE compiler, which follows the same configuration of the re-targetable ILP compiler explained in Figure 5. The front-end of the TCE compiler is the Low Level Virtual Machine (LLVM) C front-end, which transforms an application written in C to LLVM byte-code. This LLVM byte-code, known as IR, is an architecture-independent intermediate program representation used in the LLVM framework [5].

Then this IR is optimized in the middle-end and simulated with the LLI for verification. The back-end of the TCE compiler requires the architecture definition file of the target processor. In this stage, the LLVMback-end performs machine-dependent code transformations like instruction selection and register selection. After passing this stage, the optimized code contains both machine independent and dependent information. Then this optimized code is applied to the input of the TCE back-end. The back-end performs instruction scheduling, applies TTA specific optimizations, and executes the code generation process. In this paper, we show this optimized code generated for a custom CRC architecture.

Ⅳ. Simulation Results

In previous sections, the compiler characteristics are discussed elaborately. We already developed many algorithms in information and communication areas.[11-12] But, in this section, we will explain the generated TTA instructions using efficient custom target architecture for CRC implementation. In this paper, a custom ADF file named crcfast. adf is created by adding a custom FU known as CRCFAST to the minimal architecture shown in figure 1.

At first, we will show the simulation result using Tensilica processor andcompare the result with TTA processor. Finally, we will show the generated code using TTA processor.

To compile an application in XX, we required to inform Xplorer project to compile the processor configuration to compile the project on and the build target. A set of build properties like compiler, assembler and linker contains in a build target.

In this work, we took the "release" version of the target library using level-3 optimization and apply FLIX & TIE instructions. Now we are compiling the CRC 32 reference code along with its library for each of the sixteen target cores and then run a profile execution.

<Table 1> CRC32 profile status on different processor configurations

Active Processor Configuration	Total cycles	Active Processor Configuration	Total cycles
DC_C_106 micro	1987	DC_D_212GP	7054
DC_C_108 mini	1940	DC_D_233L	10804
DC_C_212GP	7050	DC_D_330 HiFi	3200
DC_C_233L	10,830	DC_D_545CK	1601
DC_C_330 HiFi	3195	DC_D_570T	6,022
DC_C_545CK	1604	XRC_D2MR	2,744
DC_C_570T	6017	XRC_D2SA	1,594
DC_D_106 micro	1990	XRC_D2SA_TIE	1,572
DC_D_108 mini	1944	XRC_D2SA_FLIX	1267

Table 1 shows the result of all target processor in terms of cycles. We can see that, without custom instruction operation XRC_D2SA is the best in comparison to other processors. Moreover, in Diamond controller processor, 545CK configuration outperforms compare to others. We see that, 545CK processor contains many DSP instruction extensions and SIMD execution units. If we see the disassembly information of input function, it is easily possible to find the step by step cycle consumptions by main and children functions as per their configuration details. We are not going to discuss all these architectural analysis.

As mentioned before, ConnX D2 architecture is suitable for communication and for its rich hardware resources, XRC_D2SA configuration without TIE or FLIX instruction, takes 1594 total cycles for CRC 32 application.

<Table 2> Profile of different funcions in CRC 32 for XRC_D2MR configurations

Active Processor Configuration	Cycle Consumption		
XRC_D2SA	Main	Reflect	CRCFAST
	1026	841	1026
XRC_D2SA_FLIX	715	534	702

Table 2 shows the cycle status consumed by different functions of CRC 32 input application. Therefore, from its profile status, CRCFAST custom function consumes highest 1026 cycles and if we see the disassembly profile of CRCFAST function, it takes many load, add, move and logical operations. So, when we think in terms of hardware, these operations are rewiring certain bits from input to output. For this reason, we develop TIE and FLIX instructions and include these custom instructions to the processor. Significant improvement in terms of cycle counts was found from table 1 and table 2, the XRC_D2SA_FLIX configuration took only 1267 cycles and CRCFAST took only 702 cycles which is almost 32% improvement compared to without FLIX operation.

Now we will compare the performance results of XX and TCE tools. It is necessary to mention that we already developed different configuration architectures using TCE tool and these architecture developing procedures are discussed in ref [4]. Like TIE and FLIX instructions in Xtensa, in TCE, we can add custom function unit to basic architecture. As a result, processor took less cycles to execute the input application. In TCE tool, there are some other techniques like modify the RFs, include more FUs by observing the cycle count of ALU& LSU, increase the number of bus etc to improve the performance of architectures those are mentioned in [4].

<Table 3> Comparison of cycle counts for the TCE and Tensilica processors

TCE			Tensilica	
Architecture Name		Cycle Count	Cycle Count	Architecture Name
minimal.adf	Bus 1	5031	2744	XRC_D2MR
	Bus 2	2244		
custom.adf		958	1601	DC_D_545CK
custom_1.adf		829	1572	XRC_D2SA_TIE
crcfast.adf	Bus 1	15	1267	XRC_D2SA_FLIX
	Bus 2	7		

Table 3 shows the comparative result between two processor tools. Under TCE tool, minimal. adf is the minimum structure architecture and is simulated for single and double bus condition. Based on the resource utilization, minimal. adf, custom. adf, custom_1. adf and crcfast. adf are generated including RF unit, custom FU etc. Form table 3, it can be shown that, for bus number 2, crcfast. adf took only 7 cycles to implement this input application.

Similarly, in XX, some custom operations like FLIX and TIE instructions are used to improve the performance of processor cores. From the table 3, XRC_D2SA_FLIX architecture takes 1267 cycles, which is costly in terms of cycle count compared to TCE processor. In TTA processor, no extra cycles are required for executing the operation of the instructions. It is occurred as the side effect of data transport.

Operation definitions of processors designed with TCE are stored in a database called the Operation Set Abstraction Layer (OSAL) [3]. OSAL stores the simulation behavior of each operation. For designing a custom FU like CRCFAST, we need to write its operation in C++ and compile to plug-in modules, which can be linked dynamically to runtime simulator.

In TCE, custom operations can be applied by using a tool called Operation Set Editor (OSEd). This OSEd is a graphical user interface to edit and debug the OSAL operation definitions. The following table shows the TTA programs generated for using custom FU CRCFAST.

Table 4shows the TCE instructions of the CRC implementation using the crcfast. adf structure. By using two buses, the parallel executions are executed and the total cycle count is dropped to 7. These seven executions are mentioned in table 4. In cycle 5, the output of CRCFAST FU is transferred to the input LSU through Bus No. 2. If there is one bus to simulate the crcfast architecture, this parallel execution is going to become sequential executions and take more cycle counts. Parallel execution depends on the number of independent operations, otherwise it takes operation latency and the instruction compiler must also take operation latencies into account.

<Table 4> TCE assembly instructions for CRC implementation with crcfast.adf

Cycle	Bus 1	Bus 2
1	4 -> ALU.in2,	16777208 -> ALU.in1t.sub ;
2	0 -> CRCFAST.trigger.crcfast,	ALU.out1 -> RF.0 ;
3	gcu.ra -> LSU.in2,	_exit ->gcu.pc.call ;
4	ALU.out1 -> LSU.in1t.stw,	... ;
5	8 -> LSU.in1t.stw,	CRCFAST.output1 ->LSU.in2;
6	...,	... ;
7	0 -> LSU.in2,	4 -> LSU.in1t.stq ;

Ⅴ. Conclusions

At first, we profile our input application using different configuration cores under Tensilica tool. Then, we discuss about the custom level operation and apply TIE and FLIX instructions to ConnX D2 core for getting improved performance. Besides it, this paper represents a framework that is used to generate the code information of TTA architecture from unscheduled HLL to scheduled TTA instructions for performing ILP. We discuss this framework as a part of the compiler back-end in TCE. This framework takes two inputs: the first is the generic byte-code of unscheduled source applications translated by a compiler front-end; the second one is a custom function architecture named crcfast. adf. Then, this framework transforms the user-defined code and scheduling chain, optimizes the scheduler, and compiles the source program to a form executable for custom target architecture. As a result, the parallel instructions, discussed in simulation results section, are formed. By using these two buses, this custom target architecture (crcfast. adf) takes 7 cycles to implement a CRC application. Moreover, from TCE tool, we took the performances of some architecture to compare the result of Tensilica configurations and found a remarkable judgment from this assessment. So we think by using this way it is possible to generate an efficient, application-specific processor. This method can be applicable many channel codes and source codes.

참고문헌

[1] C. Hendrik, "Transport Triggered Architectures Design and Evaluation," Ph. D Dissertation, Technical University of Delft Standford University, Delft, Netherlands, 1995.

[2] P. Jaaskelainen, "From Parallel Programs to Customized Parallel Processors," Ph. D Dissertation, Tampere University of Technology, Tampere, Finland, 2012.

[3] P. Jaaskelainen, "Instruction Set Simulator for Transport Triggered Architectures," Master Dissertation, Tampere University of Technology, Tampere, Finland, 2005.

[4] Alam S. Shamsul, Choi GoangSeog, "Response of Transport Triggered Architectures for High-speed Processor Design," IEICE Electronics Express Vol. 10, No. 5, 2013, pp. 1-6.

[5] Metsahalme, "Instruction Scheduler Framework for Transport Triggered Architectures," Master Dissertation, Tampere University of Technology, Tampere, Finland, 2008.

[6] P. Jaaskelainen, V. Guzma, A. Cilio, and J. Takala, "Codesign Toolset for Application-Specific Instruction-Set Processors," Proceedings of the Conference on Multimedia on Mobile Devices, SPIE, Nov 2007.

[7] Tensilica. com, ConnX D2 DSP Engine. http://www.tensilica.com/uploads/pdf/connx_d2_pb.pdf, 2012.

[8] Tensilica. com, ConnX D2 DSP Engine, http://www.tensilica.com/uploads/pdf/HiFi_2_product_brief.pdf, 2012.

[9] Tensilica Diamond Standard Controller Data Book. http://www.tensilica.com/products/diamonds#designtools, 2012.

[10] Tensilica. com, Xtensa 7 Product Brief. http://www.tensilica.com/uploads/pdf/xtensa_7.pdf, 2012.

[11] 하산타릭, 최광석, "효율적인 정도 생성기 및 새로운 순열 기법을 가진 LT 코덱구조," 디지털산업정보학회 논문지, 제10권, 제4호, 2014, pp. 117-125.

[12] 무하마드 아심, 최광석, "무선채널에서 결합 분수 부호들의 성취율 평가," 디지털산업정보학회 논문지, 제8권, 제1호, 2012, pp. 147-155.

■ 저자소개 ■

알람 삼술
(Alam S. Shamsul)

2013년 9월~현재
Khulna 대학교 전자통신공학과 조교수
2013년 8월 조선대학교 정보통신공학과 (공학석사)
2003년 2월 Khulna 대학교 전자통신공학과 (공학사)

관심분야 : 통신 VLSI, ASIC, ASIP 설계, 채널 부호
E-mail : alam_ece@yahoo.com

최 광 석
(Choi Goangseogg)

2006년 3월~현재
조선대학교 정보통신공학과 교수
2002년 2월 고려대학교 전자공학과(공학박사)
1989년 2월 부산대학교 전자공학과(공학석사)
1987년 2월 부산대학교 전자공학과(공학사)

관심분야 : 통신 및 디지털 미디어 SoC 설계
E-mail : gschoigs@chosun.ac.kr

논문접수일: 2015년 7월 16일
수 정 일: 2015년 8월 3일
게재확정일: 2015년 8월 21일