Volume 3, No. 2, February 2012

Journal of Global Research in Computer Science

ISSN-2229-371X

RESEARCH PAPER

Available Online at www.jgrcs.info

DEVELOPING A RETARGETABLE COMPILER FOR MIPS32K AND ARM7TDMI-S

Dr. Manoj Kumar Jain*1, Veena Ramnani2

*1,2Department of Computer Science, Mohanlal Sukhadia University, Udaipur, India

manoj@cse.iitd.ernet.in,

ramnaniv@yahoo.com

Abstract: The market of embedded systems is spreading faster than that of information technology. Mostly, the segments of embedded systems are consumer markets, with very short product lifetimes and short market windows. Hence, time-to-market is an important factor. Cutting down the time to market for products that became more and more complex is possible through "re-use". Another important characteristics of the embedded system market is the ease of incorporating late design changes, i.e. flexibility of the target technology, This led to the use of processors in embedded systems. This in turn led to the use of embedded software. Traditional compiler technologies were not adequate for applications and architectures of embedded systems; this led to the development of "retargetable compilers". A compiler is said to be retargetable, if it can be applied to a range of different target processors, by re-using most of the code. This means that target model cannot be an implicit part, but must be specified explicitly. In this paper, we have described the development of a retargetable compiler. The developed methodology has been used to generate and validate codes for MIPS and ARM processors. The objective of this research is to develop a retargetable compiler that can generate efficient code in terms of code size, cycle count and retargetability efforts for a wide rnge of processors.

Keywords: ASIP; Design Space Exploration; Retargetable Compilers; Register Allocation; Instruction Scheduling;

INTRODUCTION

Modern system-level design libraries frequently consists of Intellectual Property (IP) blocks such as processor cores that span a spectrum of architectural styles, ranging from traditional DSPs and superscalar RISC, to VLIWs and hybrid ASIPs. Embedded systems facilitate easy re-design of processor-memory based systems. The designer can incorporate modifications in the behavior and operation aspect of the architecture late in the design stage. ASIP are a compromise between the non-programmable ASICs and general purpose processors (GPP).

ASIP design [1][2][3][4] allows a wide range of memory organizations and hierarchies to be explored and customized for the specific embedded application. The ASIP designer is faced with the task of rapidly exploring and evaluating different architectural and memory configurations. Furthermore, shrinking time-to-market has created an urgent need to automatically generate compiler/simulator tool-kit. There are two approaches for performance estimation using ASIP design: scheduler based approach and simulator based approach.

Scheduler Based Approach:

In scheduler based approach the problem is formulated as a resource constrained scheduling problem. The application is scheduled to generate an estimate of the cycle count.

Simulator Based Approach:

A retargetable compiler is constructed for each architecture to be evaluated. This compiler is used to generate code. This generated code is given as input to a retargetable simulator which is also designed for the same architecture under evaluation. This simulator generates the performance estimates and other statistics.

RETARGETABLE COMPILERS

Retargetable compilers are a promising approach for automatic compiler generation. A compiler is said to be 'retargetable' if it can be used to generate code for different processor architectures by reusing significant compiler source code. This has resulted in a paradigm shift towards a language-based design methodology using Architecture Description Language (ADL) for embedded System-on-Chip (SOC) optimization, exploration of architecture /compiler co-designs and automatic compiler/simulator generation. However, whatever approach is used, the performance depends on the back end of the compiler i.e. instruction selection, register allocation and instruction scheduling.

In this paper, we have discussed developing a retargetable compiler which can generate code for MIPS architecture. We have divided the description under the following heads: development of Lexical Analyzer (Scanner), Development of Syntax Analyzer (Parser), and Development of Back end.

Development of Lexical Analyzer:

The purpose of lexical analyser (Scanner) is to separate the input file into logical units called tokens. The input file is a C program for which we wish to generate assembly code. The tokens in a C program can be keywords, constants (Numeric –real and integer, string, character), variables, operators, punctuation marks, etc. The lexical analyzer chooses the tokens according to a prioritized list. Normally, the order in which tokens are defined in the input to the lexical analyzer indicates priority (earlier defined tokens take precedence over later defined tokens). Hence, keywords have been defined before variable names, which means that, for example, the string "if" is recognized as a keyword and not a variable name.

The longest prefix of the input that matches any token is chosen. The principle of the longest match takes precedence over the order of definition of tokens. The principle of the longest matching prefix is implemented in the program. The tokens specified in the program are in the following order:

a. Operators (Arithmetic + , - , * , / , Assignment , Relational < , <= ,> , >= , == ,!= , Increment ++ , Decrement **--**)
b. Punctuation Marks ([,] ,) , (, { , } , ; ,”,”)
c. Integer Constants
d. Floating Point Constants
e. Keywords (int , long , float , for , if ,then ,else , continue , break , goto, while ,do, etc)
f. Variable Names

The tokens are represented by enclosing them in angle brackets < > . For e.g.

a. Arithmetic Operators are represented as <+> , <- > , <*> , < / > , <= >
b. Increment and Decrement Operators as <INR> and <DCR> respectively.
c. Relational Operators as <EQ>, <NE>, <GT>, <GE>, <LT>, <LE>.
d. Punctuation Marks as <[> , <]> , <)> , <(> , <;> , < , >,etc.
e. Keywords as <INT> , <FOR> , <IF> , <ELSE>,etc.

The integer constants, floating point constants and variable names are represented in slightly different manner. Variable names are represented as <ID, variable name> and integer and floating constants are written as <NUM, constant>.

Development of Synatx Analyser (Parser):

While lexical analysis splits the input into tokens, the purpose of syntax analysis (parsing) is to combine these tokens into a syntactic structure called the syntax tree. As the name indicates, this is a tree structure. The leaves of this tree are the tokens found by the lexical analysis. The syntax tree typically represents different program constructs like expressions, if statement, for statement, etc.

The grammar used in the proposed Compiler handles expressions and different statements like the “if” statement, “while” statement , “do” statement, “for” statement , “break” , “continue”.The productions are as below :

Stmts→stmt stmts | ϵ
Stmt → loc = bool
　　| **if** (bool) stmts
　　| **if** (bool) stmts **else** stmts
　　| **while** (bool) stmts
　　| **do** stmts **while** (bool)
　　| **for** (init_st ; bool ; update_st) stmts
　　| **break**;
　　| **continue**;
Stmt → loc = B
loc → id loc’
loc’→ [bool] loc’ | ϵ
B → J B’
B’ → || J B’ | ϵ
J → R J’
J’ → && R J’ | ϵ
R → E < E | E <= E | E > E | E>=E | E==E | E != E
E → T E’
E’ → + T E’ | -T E’ | ϵ
T →F T’
T’ → *F T’ | / F T’ | ϵ
F → **id** | **NUM** | **TRUE** | **FALSE** | **-** F | **!**F | (B)

The above grammar belons to a class of LL(1) and we have used predictive parser. Predictive parser is a top down parser constructed for LL1(1) grammar. This parser scans the input from left to right producing the leftmost derivation using one input symbol of lookahead at each step to make parsing decisions. As can be seen above , the LL(1) grammar is rich enough to cover most programming constructs. Predictive parser selects a proper production for a non-terminal by looking at the current input symbol.

Syntax-Directed Translation:

A syntax-directed definition is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. We associate information with a language construct by attaching “attributes” to the grammar symbols. Rules or semantic actions are enclosed withincurly brackets. The position of a semantic action in a production dtermines the order in which the action is executed. In our compiler, we have done translation during parsing ,without building an explicit tree. We first build the syntax tree and then convert it to 3-address intermediate representation.The syntax-directed translation for different programming constructs in C is given in the table 1:

Development of Back End of the Compiler:

The back end of the compiler is the most crucial one , it is concerned with generating machine code. The generation encompasses of instruction selection , register allocation and scheduling. We shall cover them one by one. In the proposed retargetable compiler , we intend to generate code for MIPS 32 K . We have considered the standard instruction set of both these architectures.

Instruction Selection:

The proposed compiler generates machine code for MIPS 32 K and ARM. The standard instruction set of the two processors in considered. The details of MIPS 32 instruction set has been refereed from [5] [6] [7].

Table 1. Syntax-directed translation for c constructs

Production	**Semantic rules**
Stmt → loc = bool	Stmt.value=(loc.value || “=” || bool.value)

Stmt → **if** (bool) stmts	Bool.true=newlabel()
	Bool.false=stmts.next
	Stmt.code=bool.code || label(bool.true) || stmt.code || label(bool.false)

Stmt → **if** (bool) stmts1 **else** stmts2	Bool.true=newlabel()
	Bool.false=newlabel()
	Stmts1.next = stmts2.next = stmt.next
	Stmt.code = bool.code ||
	Label (bool.true) || stmts1.code
	Gen('goto' stmt.next)
	Label(bool.false) || stmts2.code

Stmt →**while** (bool) stmts1	Start=newlabel()
	Bool.true=newlabel()
	Bool.false=stmt.next
	Stmts1.next=start
	Stmt.code=label(start) || bool.code
	|| label(bool.true) || stmts1.code
	|| gen('goto' start)

Stmt →**do** stmts1 **while** (bool)	Start=newlabel()
	Bool.true=start
	Bool.false=stmt.next
	Stmts1.next=start
	Stmt.code=label(start) || stmts1.code
	|| bool.code
	|| gen('goto' start)
	|| label(bool.false)

Stmt → **for** (init_st ; bool ; update_st) stmts1	Bool.true=newlabel()
	Bool.false=stmt.next
	Stmts1.next=stmt.next
	Stmt.code= gen(init_st)
	|| label(bool.true) || bool.code
	|| stmts1.code || gen(update_st)
	|| gen('goto' bool.true) || label(bool.false)

Register Allocation:

We have used a variation of linear scan algorithm proposed by Poletto in [8]. Our algorithm is not based on graph colouring . Rather, given the live ranges of variables in a function, the algorithm scans all the live ranges in a single pass, allocating registers to variables in a greedy fashion. The original algorithm is used for global allocation, but we have employed the same methodology for local allocation as well, taking care of the variables which are being used across the block. The formal algorithm for the above methodology is given below :

```
Procedure last_use_info()
For every basic block do
        For every variable/constant/temporary do
          Begin
            Calculate the last use in basic block
                Calculate the last use in program
          End
End Procedure
Procedure get_free_register()
              For all the registers in the register_file do
              Begin
                  If the register is holding a value which is dead,mark it as "empty"
                  If the register is holding a value that will not be used in the current block
                  But will be required globally, do not mark it as "empty"
              End
   Return the empty register
End Procedure
Procedure register_allocator()
     Call last_use_info
      Initialize all registers to "empty"
       For every basic block do
        begin
          For every variable/constant/temporary do
              Begin
                           Reg=Get_free_register ()
                           If there is a free register then
                           Allocate reg to the variable/constant/temporary
                           else
                              Find  registers  holding values which will not be used in the block
                              If no such register is found then
                                  Select the one with least usage count
                                  Spill the value in the selected register
                             End if
                           End if
                  End
               At the end of the basic block :
                   Free the registers , which are holding values which are dead
        End
End Procedure
```

Instruction Scheduling

The problem facing an instruction scheduler is to reorder machine code *instructions* to minimize the total number of *cycles* required to execute a particular instruction sequence. Unfortunately, sequential code executing on a pipelined processor inherently contains *dependencies* between some instructions. Any transformations performed during instruction scheduling must preserve these dependencies in order to maintain the logic of the code being scheduled. In addition, instruction schedulers often have a secondary goal of minimizing register lifetimes.

The Revised List Scheduling Algorithm:

We in our approach we have used a variation of list scheduling algorithm, in the sense we have combined register allocation along with list scheduling. First, the dpg(data precedence graph) is built , each instruction is a node and the data dependency between instructions is shown by drawing edges between them. Next, priorities are assigned to each node in the graph.

The formula below shows how the priority of a node is calculated:

$$\text{priority}(n) = \begin{cases} \text{latency}(n) & \text{if n is a leaf.} \\ \max(\text{latency}(n) + \max_{(m,n)\epsilon E}(\text{priority}(m)), & \text{otherwise} \\ \max_{(m,n)\epsilon E}(\text{priority}(m))) & \end{cases}$$

Input: Data Precedence Graph (DPG) with priorities assigned to each node
Output: A schedule containing all nodes in the graph that satisfies the precedence constraints in the DPG and the resource constraints of the machine
Algorithm:

```
Cycle = 1
Ready = Leaves of DPG
Active = ϕ
While (Ready U Active <> ϕ)
{
        For op= (all nodes in Ready in descending priority order)
                If (a functional unit exists for 'op' to start at 'cycle')
                {
                        -remove 'op' from Ready and add 'op' to Active
                        - add 'op' to schedule at time 'cycle'
                        - make operands available in registers and allocate a register for target
                }
        End for
        Cycle = cycle +1
        Update the Ready Queue
}
For op= (all nodes in Active)
        If ('op' finishes at time 'cycle')
        {
                -remove 'op' from Active
                - Check nodes waiting for 'op' in DPG and add to 'ready' – if all operand are available
        }
End for
```

PERFORMANCE ESTIMATES AND VALIDATION OF THE COMPILER

Validation of MIPS Code:

In order to validate the MIPS code, we have used MARS (**M**IPS **A**ssembler and **R**untime **S**imulator). MARS [9] is an Education- Oriented MIPS Assembly Language Simulator, developed by University of Missouri State. MARS is an Integrated Development Environment (IDE) controlled by a modern GUI. The performance of the generated code is judged on the basis of Code Size, Cycle Count and compilation time.

Comparison of Code Size of code generated by our Compiler and that of EXPRESS:

The length of the code is an important quality metric. It is important that the size of the code generated by the compiler is kept minimal. We have compared the code generated by our compiler with the one generated by GCC cross compiler and the code generated by EXPRESS. The GCC cross compiler can generate machine code for various platforms. We have configured GCC to generate code for MIPS and ARM. We observe that our code is much smaller than the code generated by GCC cross compiler for MIPS and ARM code and it is comparable to the size of the code generated by EXPRESS for MIPS code.

Comparison of Cycle Count of code generated by our Compiler and that of EXPRESS:

The performance of the MIPS code has been tested against the code generated by standard tools. The cycle count is an important parameter to see how fast the assembly code is. We have tested the code in terms of cycle count and found that our compiler is generating better code.

Since, we had based our research on the EXPRESS compiler and we already have the cycle counts for the MIPS code of the benchmarks. The cycle count for MIPS has been calculated using SIM-A simulator, which is capable of simulating the MIPS code. The table 2 gives the comparison of results as obtained from SIMPRESS simulator and SIM-A simulator. The results are shown graphically in the Figure 1.

Table 2: Validation Results

Benchmarks	Cycle Count for EXPRESS code (using SIMPRESS)	Cycle Count for code generated by our compiler (using Sim-A)
LL1	675	637
LL5	559	289
LL11	410	309
LL12	552	471
LL24	749	711

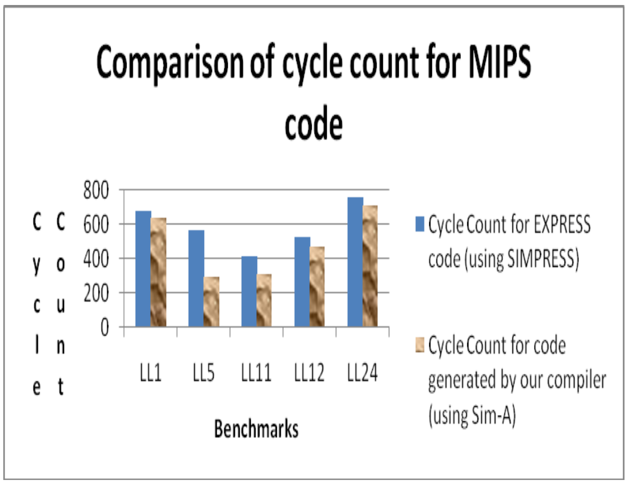

Figure 1 : Cycle Count Comparison for MIPS code

Similarly, the cycle count for ARM code has been calculated using Keil µvision4. We have simulated the C Program and calculated the cycle count and repeated the same for the ARM assembly code generated by our compiler. The cycle counts obtained from Keil are shown in Table 3: The results are shown graphically in the Figure 2.

Table 3: Keil Simulation Results

Benchmarks	C Program	ARM Assembly Code
LL1	411	400
LL5	343	338
LL11	240	238
LL12	273	258
LL24	411	400

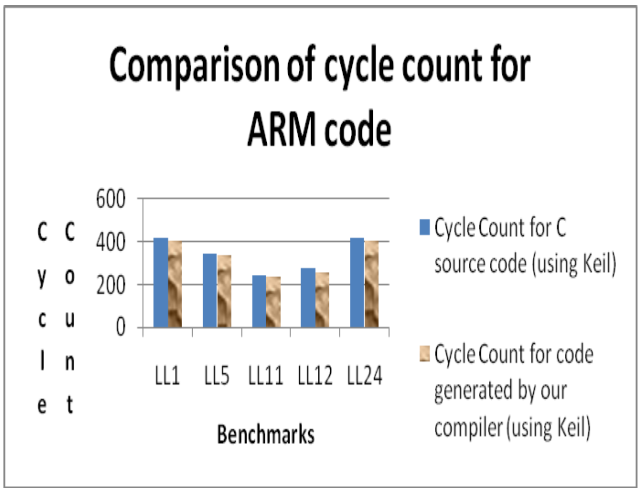

Figure 2 : Cycle Count Comparison for ARM code

We can conclude from the above results that the MIPS and ARM code generated by our compiler is better than the existing tools in terms of cycle count.

EFFICIENCY OF THE PROPOSED RETARGETABLE COMPILER

We have used compilation time as the criteria to measure performance. The compilation time basically tells us how fast or slow the compiler is. Since, we had started the research with the study of EXPRESS and EXPRESSION. We shall be comparing the compilation time results of MIPS code with those of EXPRESS. The timing from EXPRESS and our compiler are given in the Table 4. The results are graphically displayed in Figure 3.

Table 4: Compilation Time Comparison

Benchmarks	EXPRESS	Our Compiler
LL1.c	0.8 secs	0.3 secs
LL5.c	0.5 secs	0.2 secs
LL11.c	0.4 secs	0.2 secs
LL12.c	0.5 secs	0.3 secs
LL22.c	1.3 secs	0.3 secs
LL24.c	0.7 secs	0.2 secs
Compress.c	2.1 secs	1.6 secs

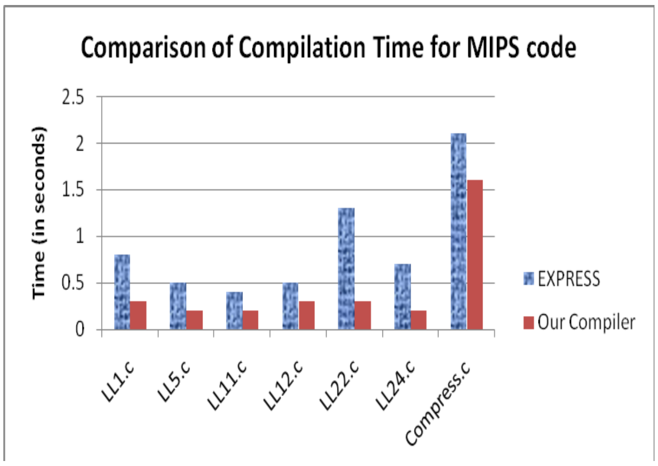

Figure 3 : Comparison of compilation time of MIPS code by EXPRESS and our compiler

From the above results, we conclude that overall the compiler developed by us takes less compilation time and hence is better than the existing EXPRESS compiler.

CONCLUSIONS

We have developed a retargetable compiler in Visual Basic. It is capable of generating MIPS and ARM code. Our compiler is a user retargetable compiler. The retargetable efforts are intermediate. Some of the information is entered as parameters through the graphical user interface and rest is used at the time of coding. The following can be provided to the CPU: size of the register file, name of registers and details of functional units. It is observed that the code is good in terms of code size, cycle count and compilation times.

REFERENCES

[1] Jain,M.K., Kumar,A., Balakrishnan,M. and Gangwar,A.(2005) Customizing Embedded Processors for Specific Applications, In proceedings of Recent Trends in Practice and Theory of Information Technology, Proc. of NRB Seminar, 10-11 January 2005, NPOL, Cochin, pp. 261-284

[2] Jain,M.K., Balakrishnan,M.and Kumar,A.(2001) ASIP Design Methodologies: Survey and Issues, In proceedings of the Fourteenth International Conference on VLSI Design, 2001, 3-7 Jan. 2001, Pages: 76-81

[3] Jain,M.K., Balakrishnan,M. and Kumar A.(2004), Efficient Technique for Exploring Register File Size in ASIP Design', IEEE TCAD of VLSI, vol. 23, No. 12, pp. 1693-1699, Dec. 2004.

[4] Jain,M.K. and Gaur,D.(2011)ASIP Design Space Exploration :Survey and Issues ,International Journal of Computer Science and Information Security,ISSN – 19475500,Volume 9, Issue 4 ,pg. 141-145

[5] MIPS 32 Architecture for Programmers (2008) – Volume I: Introduction to MIPS32 Architecture, Document Number: MD00082, Revision 2.60

[6] MIPS 32 Architecture for Programmers (2009) – Volume II: Introduction to MIPS32 Architecture, Document Number: MD00086, Revision 2.62

[7] MIPS 32 Architecture for Programmers (2009) – Volume III: Introduction to MIPS32 Architecture, Document Number: MD00090, Revision 2.80

[8] M. Poletto, D.R. Engler and M.F.Kaashoek (1997). tcc: A system for fast, flexible, and high-level dynamic code generation. In Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation. Las Vegas, NV, 109-121.

[9] K.Vollmar and P.Sanderson (2006): MARS An Education-Oriented MIPS Assembly Language Simulator, SIGCSE’06, March 1-5, 2006, Houston, Texas, USA.

Short Bio Data for the Authors

Dr.M.K. Jain received the M.Sc. degree from M.L. Sukhadia University, Udaipur, India, in 1989. He received M.Tech. degree in Computer Applications and PhD in Computer Science & Engineering from IIT Delhi, India in 1993 and 2004 respectively. He is Associate Professor in Computer Science at M.L. Sukhadia University Udaipur. His current research interests include application specific instruction set processor design, wireless sensor networks, semantic web and embedded systems.

Veena Ramnani is a research scholar in the Department of Computer Science at Mohanlal Sukhadia University, Udaipur, India. Her area of research is embedded systems design and retargetable compilers.