Volume 3, No. 12, December 2012
Journal of Global Research in Computer Science
RESEARCH PAPER
Available Online at www.jgrcs.info
ISSN-2229-371X

# VERYIFYING AND SUBSTANTIATING FUNCTIONS IN MIPS

Dr. Manoj Kumar Jain*[1], Ms. Veena Ramnani[2]
*[1] Associate Professor, Department of Computer Science, Mohan Lal Sukhadia, Udaipur (Raj.)
manoj@cse.iitd.ernet.in
[2]Research Scholar, Department of Computer Science, Mohan Lal Sukhadia, Udaipur (Raj.)
ramnaniv@yahoo.com

***Abstract***: Complexity of embedded system design is increasing day by day. An embedded system is a combination of hardware and software. Embedded systems are broadly defined as systems designed for a particular application while meeting strict design constraint. In addition, market competition and the increasing demand for electronic equipment are pushing designers to shorten the design cycles of new products. There is a well known tradeoff between retargetability and code quality in terms of performance and code size when compared to the hand optimized code. This is because when the design space is large, all possible target specific optimizations cannot be performed in that case**.** In this paper we have demonstrated the importance of retargetable compiler.
The major contribution of this paper lies in design and development of retargetable compiler for MIPS, especially implementing functions.

***Key Words:*** MIPS, Design Space Exploration (DSE), Retargetable Compiler, ASIC

## INTRODUCTION

High-tech systems ranging from smart phones to printers, from cars to radar systems, and from satellites to medical imaging equipment contain an embedded electronic core that typically integrates a heterogeneous mix of hardware and software components. The resulting platform is often distributed, and it typically needs to support a mix of data-intensive computational tasks with event processing control components. These embedded components more and more often have to operate in a dynamic and interactive environment. Moreover, not only functional correctness is important, but also quantitative properties related to timeliness, quality-of-service, resource usage, and energy consumption. The complexity of today's embedded systems and their development trajectories is thus increasing rapidly. At the same time, development trajectories are expected to produce high-quality and cost-effective products.

A common challenge in development trajectories is the need to explore extremely large design spaces, involving multiple metrics of interest (timing, resource usage, energy usage, and cost). The number of design parameters (number and type of processing cores, sizes and organization of memories, interconnect, scheduling and arbitration policies) is typically very large and the relation between parameter settings and design choices on the one hand and metrics of interest on the other hand is often difficult to determine. Given these observations, embedded-system design trajectories require a systematic approach that is automated as far as possible.

## DESIGN SPACE EXPLORATION

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] allow 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.
The ASIP design has the following steps:

### *Application Analysis:*

The application written in HLL is analyzed to find out parameters like data types used, execution count of operators and functions, life time of variables etc.

### *Architecture Design space exploration:*

The performance of various architectures is estimated and a suitable architecture satisfying the design constraints is selected. There are two approaches for performance estimation:

a) Scheduler Based Approach: In scheduler based approach as shown in Figure 3, the problem is formulated as a resource constrained scheduling problem. The application is scheduled to generate an estimate of the cycle count.

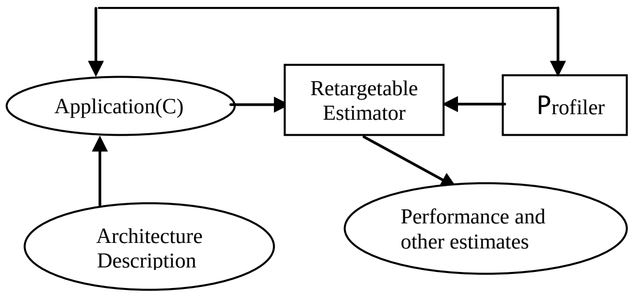

Figure 1: Scheduler based Approach

b) Simulator Based Approach: A retargetable compiler is constructed for every architecture to be evaluated. This compiler is used to generate code. As shown in figure 4, 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.

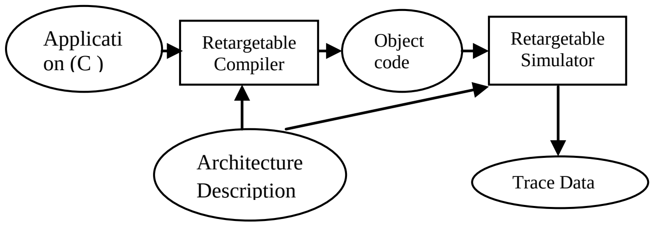

Figure 2: Simulator based approach

***Instruction Set Generation:***

The instruction set is generated for the selected target architecture.

***Code Synthesis:***

Code synthesis is done using a retargetable code Generator which will take application, the architecture template and instruction set as inputs and generate code for the target processor.

***Hardware Synthesis:***

The hardware is synthesized using the ASIP architectural template and instruction set architecture.

Scheduler based approach for design space exploration is a relatively new field, we shall be considering simulator based approach for design space exploration. 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.

## FUNCTION IMPLEMENTATION IN MIPS

Functions are perhaps the most fundamental unit of programming, used in all of programming languages. It gives us the simplest form of program abstraction. It provides an interface (i.e., the prototype) and allows us to use the function without knowing how it is implemented. Thus, it makes sense that assembly languages must provide the mechanism to implement functions. There are two ideas behind a function

a. You should be able to call the function from anywhere.
b. Once the function is complete, it should return back to the place that called the function.

Function calls are relatively simple in a high-level language, but actually involve multiple steps and instructions at the assembly level.

a. The program's flow of control must be changed.
b. Arguments and returning values are passed back and forth.
c. Local variables can be allocated and destroyed. Invoking a function changes the flow of program twice:

a) Calling the function: Every time a function is called, the CPU has to remember an appropriate return address.
b) Returning from a function: When the function execution is complete, the CPU has to restore the return address in the Program Counter.

MIPS [5] [6] [7] uses the jump-and-link instruction **jal** to call functions.

a) The jal saves the return address (the address of the next instruction) in the dedicated register $ra, before jumping to the function.
b) Jal is the only MIPS instruction that can access the value of the program counter, so it can store the return address PC+4 in $ra.

E.g. jal Fact

To transfer control back to the caller, the function just has to jump to the address that was stored in $ra.

```
jr $ra
```

Functions accept arguments and produce return values.MIPS uses the following conventions for function arguments and results.

—Up to four function arguments can be "passed" by placing them in registers **$a0-$a3** before calling the function with **jal**.

—A function can "return" up to two values by placing them in registers **$v0-$v1**, before returning via **jr**.

## FUNCTION IMPLEMENTATION IN OUR COMPILER

Our compiler is capable of generating MIPS code for simple programs [8] and now its functionality is being extended to implement functions. The register conventions of MIPS have been retained in our compiler as well. Registers $a0-$a3 are used for passing arguments to the functions, $v0-$v1 are used by the functions to return values from the functions.Regsiters $s0-$s7 are used within the functions for local variables.

When we program in C or C++ or Java, we are used to calling functions with local variables. Each time we call a function, a new set of local variables is created. This is why recursive function calls work. Each recursive call has its own copy of local variables and parameters (unless the parameters are passed by reference). This makes it easier to write functions in procedural languages.

When we program in assembly language, there is only one set of registers used in the program. In effect, these registers act like global variables. It's very easy to make a function call, and think that after the function call is done, the registers have remained unchanged. When we make a function call, we have to assume that, unless convention dictates otherwise, the function will clobber all the registers we are using (except the stack pointer). Thus, if we call a function, any values we have stored in a register *could* be overwritten. After all, the function being called needs to use registers too and there's only one set to work with.

We can keep these registers from being overwritten by a function call, by saving them before the function executes, and restoring them after the function completes. We have used the convention that at the time of function call, the compiler first checks which register have values which will be required subsequently in the following blocks and then saves them on the stack. These values are restored after the control returns from the function. This is especially important for nested functions.

The algorithm for implementing functions in our compiler is as follows:

a. Check the function registers $s0-$s7 and free the ones whose contents are dead or not live. If the contents are live i.e. have "next use" in the upcoming block, save the contents on the stack.
b. Initialize function registers $s0-$s7 to empty
c. Place the arguments in the registers $a0-$a3
d. Branch to the assembly code of the function.
e. Execute the code
f. At the end place the return values in the registers $v0-$v1
g. Branch to the return address stored in the register $ra
h. Restore the contents of function registers $s0-$s7 from the stack

Each function, as it is running, will have a part of the stack for its own use. This is called the *stack frame*. By convention, the functions just use its part of the stack. The exception is when the called needs to access arguments passed by the caller. The arguments are considered part of the caller's stack.

It may be possible for a function code to have more than one stack frame. For example, recursive functions will have a stack frame for each recursive call that's made. Functions do not need to be recursive for there to be two or more stack frames associated with the function.

## VERIFYING AND SUBSTANTIATING THE COMPILER

The assembly code generated for functions was functionally verified with the help of 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 whose features include:

a. control of execution speed, including single step at variable speed (slider bar controls the number of instructions per second)
b. thirty-two registers visible at the same time, selectable via tabbed interfaces,
c. "spreadsheet" (WYSIWYG) modification of values in registers and memory,
d. selection of data value display in decimal or hexadecimal,
e. "surfing" through memory using buttons to change display to next/previous, stack location, global partition, and the start of the memory segment,
f. an integrated editor and assembler as part of its IDE.

The quality of code can be judged by many parameters memory access operations, cache hits and misses, code size, cycle count, execution time, etc. The performance statistics we have used is the code size.

***Code size:***

The code for the benchmarks was generated using the GCC compiler [10]. The size of the code generated by our compiler is much less than that generated by GCC. A glimpse of the code generated by the two is given in the table below. The code below has been generated for initializing the array in "insertion sort".

Table: 1

| Code by Our Compiler | Code by GCC |
|---|---|
| FL1 : | fillarray: |
| li $s0,0 | addiu $sp,$sp,-16 |
| li $s1,0 | sw $fp,12($sp) |
| L2 : | move $fp,$sp |
| li $s2,10 | sw $4,16($fp) |
| bge $s1,$s2,L3 | sw $0,0($fp) |
| li $s3,4 | sw $0,4($fp) |
| mul $s4,$s1,$s3 | j $L4 |
| add $s5,$a0,$s4 | nop |
| li $s6,0 | $L5: |
| sub $s7,$s6,$s1 | lw $2,4($fp) |
| addi $s4,$s7,20 | nop |
| sw $s4,($s5) | sll $3,$2,2 |
| lw $s4,($s5) | lw $2,16($fp) |
| add $s5,$s0,$s4 | nop |
| move $s0,$s5 | addu $4,$2,$3 |
| add $s1,$s1,1 | li $3,20 |
| b L2 | # 0x14 |
| L3 : | lw $2,4($fp) |
| move $v0,$s3 | nop |
| jr $ra | subu $2,$3,$2 |
| | sw $2,0($4) |
| | lw $2,4($fp) |
| | nop |
| | sll $3,$2,2 |
| | lw $2,16($fp) |
| | nop |
| | addu $2,$2,$3 |
| | lw $3,0($2) |
| | lw $2,0($fp) |
| | nop |
| | addu $2,$2,$3 |
| | sw $2,0($fp) |
| | lw $2,4($fp) |
| | nop |
| | addiu $2,$2,1 |
| | sw $2,4($fp) |
| | $L4: |
| | lw $2,4($fp) |
| | nop |
| | slt $2,$2,10 |
| | bne $2,$0,$L5 |
| | nop |
| | lw $2,0($fp) |
| | move $sp,$fp |
| | lw $fp,12($sp) |
| | addiu $sp,$sp,16 |
| | j $31 |
| | nop |

It can be observed that the code generated by GCC contains several load/store instructions. GCC compulsorily makes use of frame pointer and stack pointer, a lot of programming effort is involved in updating these pointers and accessing memory through these pointers. As a result, the GCC generates longer assembly code as compared to that of our compiler. As far as the code generated by our compiler is concerned, we have not used the frame pointer. Also, the stack pointer is required only when we store values on the stack. Lastly, the stack is required only when there are no empty registers. We have used our own register allocation algorithm[11], which helps us achieve fewer loads and stores.

The comparison of code size in terms of number of lines generated by our compiler and GCC is given in the table below and also represented graphically in Figure 3.

Table: 2

| Benchmarks | Code Size (Our Compiler) | Code Size (GCC Compiler) |
|---|---|---|
| Matrix Multiplication | 107 | 227 |
| Insertion sort | 108 | 267 |
| Bubble Sort | 99 | 231 |
| LL1 | 66 | 172 |
| LL5 | 67 | 161 |
| LL12 | 52 | 141 |

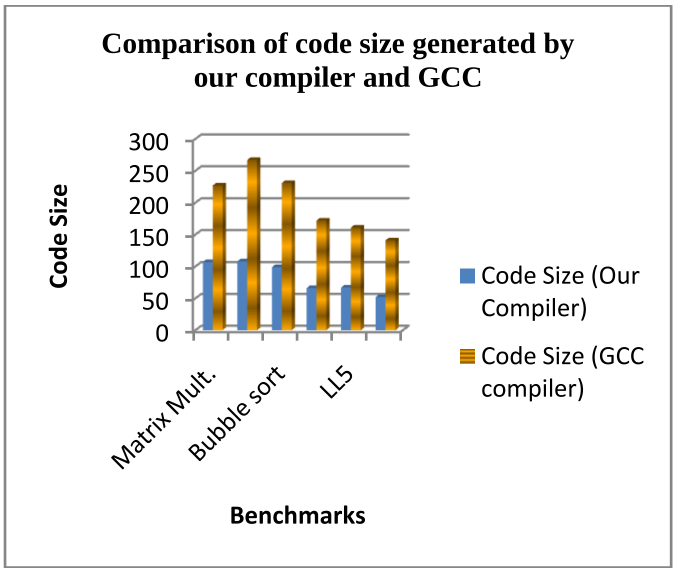

Figure 3: Comparison of code size generated by our compiler and GCC compiler.

It can be observed from the above table that the code generated by our compiler is much smaller as compared to that generated by GCC. The code size is in terms of line count.

## CONCLUSION

The functionality of an embedded system is divided into hardware and software components. Synthesis of hardware components involves designing a custom circuit for the hardware portion of input application. Synthesis of software component consists of designing a processor that is suited for the software portion of the input application and generating code that implements the functionality of the software component on the designed processor. Short design cycles and increasing embedded system complexity make it impractical to perform manual processor architecture exploration and code generation.

We have developed a retargetable compiler that can generate code for MIPS32 architecture. The compiler is capable of handling function calls and function return. The code has been functionally verified and the quality of the code has been checked for code size. We have shown that the code generated by our compiler is better than that generated by GCC compiler.

## REFERENCES

[1]. M.K. Jain, Anshul Kumar, M. Balakrishnan and Anup Gangwar: 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]. M.K. Jain, M. Balakrishnan, Anshul Kumar: 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]. M.K.Jain, M.Balakrishnan and Anshul Kumar: 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 Ramnani, V., (2007) Challenges in Retargetable Compiler Technology in ASIP Design, Indian Engineering Congress

[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.K.Jain and Veena Ramnani(2012), Developing a retargetable compiler for MIPS32k and ARM7TDMI-S, Journal of Global Research in Computer Science, Volume 3, No. 2, February 2012, ISSN-2229-371X

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

[10]. GCC Homepage: http://gcc.gnu.org

[11]. M.K.Jain and Veena Ramnani (2011), Register Allocation And Instruction Scheduling For An Efficient Retargetable Compiler, International Journal of Advanced Research in Computer Science, Volume 2, Issue 5,October 2011, ISSN No. 0976-5697